REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. REPORT DATE ***(DD-MM-YYYY)*** 20-01-2003	**2. REPORT TYPE** Final Report	**3. DATES COVERED** *(From – To)* 12-04-2002 – 12-11-2002

4. TITLE AND SUBTITLE Algorithms for Multisensor Multitarget tracking	**5a. CONTRACT NUMBER** F61775-02-WE022
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Associate Professor Leonidas S Pitsoulis	**5d. PROJECT NUMBER**
	5d. TASK NUMBER
	5e. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Technical University of Crete, Kounoupidiana, Chania 73100, Greece	**8. PERFORMING ORGANIZATION REPORT NUMBER** N/A
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) EOARD, PSC 802 BOX 14, FPO 09499-0014	**10. SPONSOR/MONITOR'S ACRONYM(S)**
	11. SPONSOR/MONITOR'S REPORT NUMBER(S) SPC 02-4022

12. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This report results from a contract tasking Technical University of Crete as follows:
1. Construction of a set of problem instances of multidimensional assignment problems in the context of target tracking. These will be used as benchmark problems. They will be constructed so that their optimal solution will be known, and they will vary in size and dimension. Furthermore they will be nontrivial to solve, since they will be used for evaluation of the proposed algorithms in the experimental runs.
2. Design and implementation of data structures to represent the massive sparse data sets associated with each instance of the problem. These data structures will be general enough to handle variable dimension and degrees of sparsity. Specific tasks to be performed by the algorithms, such as function evaluation and construction of feasible and partial solutions, should require minimum computational effort and memory.
3. Design and implementation of heuristic and exact algorithms for solving the multidimensional assignment problem. The heuristic algorithm will receive the dimension of the instance and the sparse multidimensional array as inputs, and it will provide the partitions that represent the targets. The exact algorithm will use a branch-and-bound scheme to provide exact solutions to the problem.
All the codes will be written using the C programming language.

15. SUBJECT TERMS

EOARD, Control System, Computational Mathematics, multitarget tracking

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18, NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**	UL	30	Neal D. Glassman
UNCLAS	UNCLAS	UNCLAS			**19b. TELEPHONE NUMBER** *(Include area code)* +44 (0)20 7514 4437

Research Report

ALGORITHMS FOR MULTITARGET MULTISENSOR TRACKING [*]

Leonidas Pitsoulis
Department of Industrial and Systems Engineering
Technical University of Crete
Kounoupidiana, Chania 73100, Greece

January 20, 2003

*funded by the European Office of Aerospace Research and Development grant F61775-02-WE022

Contents

1 Introduction

Multitarget Multisensor Tracking (MTMST) as the name implies, is the process of tracking a multiple of targets using a number of sensors. In general terms this process is divided into two major sub processes, one has to do with the data estimates at each discrete point in time, and the second with the development of the target trajectory based on the data estimates. The process of combining data from a number of sensors taken at a discrete point in time is called sensor fusion, and the corresponding combinatorial problem is the **data association problem**.

When tracking a single target with a single sensor, noisy measurements are made at discrete, periodic points in time from the sensor regarding the moving target. The goal is to establish a time-continuous estimate of the target's trajectory. For the MTMST problem, noisy measurements are made from an arbitrary number of spatially diverse sensors regarding an arbitrary number of targets with the goal of estimating the trajectories of all of the targets present. Furthermore, the number of targets may change (e.g., by wandering out of and into detection range) during the problem time interval and the sensors themselves may be in motion. For example in Figure 1, two sensors observe three targets, thereby we have three measurements from each sensor. The MTMST problem is to partition all six measurements (three from each of the 2 sensors) into three subsets that correspond to targets.

rag replacements

x
y
sensor 1
sensor 2
terrain
targets

Figure 1: Sensor fusion

Target state estimates for single-target single-sensor tracking are obtained satisfactorily by applying one of several variants of the Kalman filter. However, no method known can consistently obtain good trajectory estimates for the more general MTMST problem.

The MTMST has generally been divided into 2 sub problems: state estimation/propagation and data association. The combinatorial nature of the MTMST problem results from the data association sub problem; that is, given that each sensor m, $m = 1 \dots M$, makes n_m measurements, $n_i \neq n_j, \forall i \neq j$, how do we correctly partition the measurements into all of the targets, ensuring that all of the measurements are used exactly once? Total enumeration of all possible partitions is obviously unrealistic, since the size of the search space grows exponentially.

In MTMST a data association problem is presented at discrete, periodic moments in time. We expect that by obtaining the correct partition or association, the data from *all* of the sensors may be combined in some manner to make estimates of target positions that are more accurate than that obtained from a single measurement.

As it is developed in the paper by Murphey, Pardalos and Pitsoulis [9], the data association problem with respect to the MTMST can be modeled mathematically as a **Multidimensional Assignment Problem (MAP)**. In this research we will examine the following aspects of the MAP:

- Problem Generators that will produce problem instances with known optimal solutions.
- Efficient data structures to handle the massive data sets associated with the instances of the MAP.

- Heuristic algorithms for providing near optimal solutions to the MAP.
- Lower bounds and exact algorithms for solving the MAP.
- A comprehensive survey on the MAP which will report all the latest research developments for this important combinatorial optimization problem.

The objective of this research is to develop efficient algorithms for solving the multidimensional assignment problem in real time.

2 Formulations

For most combinatorial optimization problems, different mathematical formulations exist that are equivalent. They usually reflect the effort to view the structural characteristics of the problem from a different angle, in hope of developing new solution techniques. In this section we will present the various equivalent mathematical formulations for the MAP. Let us denote a multidimensional assignment problem of dimension m with MAP(m).

2.1 0-1 Integer Programming Formulation

In what follows the optimization problem formulation of the multidimensional assignment problem will be presented, first as an Integer Program (IP). Let $n_1, n_2, \ldots, n_m$ be integers which define the ranges in each dimension. The input data consists of one cost array $C \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_m}$, where without loss of generality assume that $n_1 \le n_2 \le \cdots \le n_m$. The IP formulation of the m-dimensional assignment problem is the following:

$$\mathrm{MAP}_{IP}(m): \quad \min \quad \sum_{i_1=1}^{n_1} \cdots \sum_{i_m=1}^{n_m} C_{i_1 \cdots i_m} x_{i_1 \cdots i_m} \tag{1}$$

$$\text{s.t.} \quad \sum_{i_2=1}^{n_2} \cdots \sum_{i_m=1}^{n_m} x_{i_1 \cdots i_m} = 1, \quad i_1 = 1, \ldots, n_1, \tag{2}$$

$$\sum_{i_1=1}^{n_1} \cdots \sum_{i_{k-1}=1}^{n_{k-1}} \sum_{i_{k+1}=1}^{n_{k+1}} \cdots \sum_{i_m=1}^{n_m} x_{i_1 \cdots i_m} \le 1, \tag{3}$$

$$\text{for } i_k = 1, \ldots, n_k \text{ and } k = 2, \ldots, m-1 \tag{4}$$

$$\sum_{i_1=1}^{n_1} \cdots \sum_{i_{m-1}=1}^{n_{m-1}} x_{i_1 \cdots i_m} \le 1, \quad i_m = 1, \ldots, n_m, \tag{5}$$

$$x_{i_1 \cdots i_m} \in \{0, 1\} \text{ for all } i_1 \cdots i_m. \tag{6}$$

We will call the constraints in the above formulation **multidimensional assignment constraints**. The inequalities are a direct consequence of the fact that the ranges $n_1, n_2, \ldots, n_m$ are not equal. However we could assume without loss of generality that $n_1 = n_2 = \cdots = n_m$, by introducing artificial costs in the cost array $C \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_m}$ with a very high value (thereby forcing them not to be included in the optimal solution).

Example 2.1 *Consider that we have $m = 3$, or in other words we have the well known three dimensional assignment problem where using the 0-1 formulation stated in (2)-(6) we* *will have:*

$$
\begin{aligned}
MAP_{IP}(3): \quad & \min & & \sum_{i=1}^{n_1}\sum_{j=1}^{n_2}\sum_{k=1}^{n_3} c_{ijk}x_{ijk} \\
& s.t. & & \sum_{j=1}^{n_2}\sum_{k=1}^{n_3} x_{ijk} = 1, \quad i = 1,\ldots,n_1, \\
& & & \sum_{i=1}^{n_1}\sum_{k=1}^{n_3} x_{ijk} \le 1, \quad j = 1,\ldots,n_2, \\
& & & \sum_{i=1}^{n_1}\sum_{j=1}^{n_2} x_{ijk} \le 1, \quad k = 1,\ldots,n_3, \\
& & & x_{ijk} \in \{0,1\} \;\; \forall \; i,j,k
\end{aligned}
$$

where in the above formulation we assume that $n_1 \le n_2 \le n_3$.

The above mentioned IP formulation of the MAP however is not very convenient with respect to algorithmic issues, it does however provide an insight to the nature of the problem. For example it can be used to derive lower bound for the problem by relaxing the integrality constraints and solving the linear program. It has also been used in a Lagrangian relaxation solution scheme in [13].

2.2 Permutations

In this section we will offer a different formulation for the MAP using permutations. This formulation will provide essential insight so as to construct efficient data structures for the problem and local search procedures. The main characteristics of this formulation is its compactness, and its combinatorial nature.

Let $\mathbf{X}^m$ be the set of all arrays $X = (x_{i_1\cdots i_m})_{n_1\times\cdots\times n_m}$ such that their elements satisfy the constraints (2) through (6). We will call $\mathbf{X}^m$ **permutation arrays.**

Given a set of integers $N = \{1,2,\ldots,n\}$ a partial k-permutation on N will be defined as the mapping $p : \{1,2,\ldots,k\} \to N$, where $k \le n$. We write $p := \begin{pmatrix} 1 & 2 & \cdots & k \\ p(1) & p(2) & \cdots & p(k) \end{pmatrix}$ but for short from now we will write $p = (p(1),p(2),\ldots,p(k))$. For example for $N = \{1,2,3,4,5\}$ a partial 3-permutation might be $p = (5,3,1)$. Denote the set of all k-permutations of a set of integers $N = \{1,2,\ldots,n\}$ as $\Pi_{(N,k)}$.

Remark 2.1 *Given that* $|N| = n$ *and some integer* $k = 1,2,\ldots,n$ *we have* $|\Pi_{(N,k)}| = \frac{n!}{(n-k)!}$.

Define the sets of integers $N_i := \{1,2,\ldots,n_i\}, i = 1,2,\ldots,m$, and let $\Pi_{(N_i,n_1)}$ be the set of all n_1-permutations $p_i : N_1 \to N_i$. Note that $|\Pi_{(N_i,n_1)}| = \frac{n_i!}{(n_i-n_1)!}$, for each $i = 1,\ldots,m$. Given $p_1,\ldots,p_m$ let us define a matrix $P := (p_1,\ldots,p_m)$, that is P can be regarded as a $n_1 \times m$ matrix with columns the n_1-permutations $p_1,\ldots,p_m$,

$$
P := \begin{pmatrix} p_1(1) & p_2(1) & \cdots & p_m(1) \\ p_1(2) & p_2(2) & \cdots & p_m(2) \\ \vdots & \vdots & \ddots & \vdots \\ p_1(n_1) & p_2(n_1) & \cdots & p_m(n_1) \end{pmatrix}. \tag{7}
$$

Now given $n_1 \le n_2 \le \cdots \le n_m$, and denoting by $e_{N_1} = (1, 2, \ldots, n_1)$ the identity permutation of N_1, let $\mathbf{P}^m$ be the following set

$$\mathbf{P}^m := \{P = (p_1 \ldots, p_m) \ : \ p_1 = e_{N_1}, p_i \in \Pi_{(N_i, n_1)}, i = 2, \ldots, m\}.$$

Remark 2.2 $|\mathbf{P}^m| = \prod_{i=2}^{m} \frac{n_i!}{(n_i - n_1)!}$.

Note that there is a one-to-one correspondence between the elements of $\mathbf{X}^m$ and $\mathbf{P}^m$ for given $n_1 \le n_2 \le \cdots \le n_m$, since given any $P \in \mathbf{P}^m$ we can construct an $X \in \mathbf{X}^m$ as

$$x_{i_1 \cdots i_m} = \begin{cases} 1 & \text{if } p_j(i_1) = i_j, j = 2, \ldots, m, \\ 0 & \text{otherwise,} \end{cases} \tag{8}$$

for $i_1 = 1, \ldots, n_1$, and vice versa. We can therefore formulate the MAP using permutations as follows:

$$\begin{array}{lll} \text{MAP}_\Pi(m): & \min & \displaystyle\sum_{i=1}^{n_1} C_{ip_2(i)\cdots p_{m-1}(i)p_m(i)} \\ & \text{s.t.} & p_i \in \Pi_{(N_i, n_1)}, \quad i = 2, \ldots, m. \end{array} \tag{9}$$

A solution to the MAP using the above formulation will be any $P \in \mathbf{P}^m$ such that the above objective function is minimized. Moreover from remark 2.2 we can conclude about the size of the feasible space which is finite, but of course it grows factorially with the problem size and dimension.

Example 2.2 *Consider that we have $m = 2$ with $|N_1| = n_1$ and $|N_2| = n_2$ with $n_1 \le n_2$. This is the well known linear assignment problem:*

$$\begin{array}{ll} \min & \displaystyle\sum_{i=1}^{n_1} C_{ip(i)} \\ s.t. & p \in \Pi_{(N_2, n_1)}, \end{array} \tag{10}$$

where it is well known that it can be solved in $O(n^3)$ computational time.

2.3 Graph Theoretic

Consider a three dimensional assignment problem where we have $G = (V, E)$ and $V = V_1 \cup V_2 \cup V_3$. The graph will contain edges and *hyperedges*, and the edge set will be composed as follows, $E = E_1 \cup E_2 \cup E_3 \cup E_4$ where $E_1 = V_1 \times V_2$, $E_2 = V_2 \times V_3$, $E_3 = V_1 \times V_3$ and finally $E_4 = V_1 \times V_2 \times V_3$. We will denote an element of E by e_{ij} if it is the edge joining vertices i and j, or e_{ijk} if it is the hyperedge joining the vertices i, j and k. An instance of such a three-partite graph with a feasible solution is given in Figure 2. Based on the above example we can now formulate the multidimensional assignment problem as the following problem on graphs:

MAP$_G(m)$: *Given an m-partite graph $G = (V, E)$ with vertex set: $V = V_1 \cup V_2 \cup \cdots \cup V_m$ were $|V_1| \le |V_2| \le \cdots \le |V_m|$, and a set of hyperedges $E \subseteq V_1 \times V_2 \times \cdots \times V_m$, where equality holds when the problem is dense, and a cost function $c : E \to \mathbb{R}$, find $|V_1|$* **disjoint cliques of size** *m with minimum sum of edge costs.*

PSfrag replacements

i
j
k
e_{ij}
e_{ik}
e_{jk}
e_{ijk}
V_1
V_2
V_3

Figure 2: A tri-partite graph

2.4 Inner Product

This formulation of the MAP is an immediate result from the integer programming formulation of the problem, by making the following generalization of the inner product. More specifically, let the inner product between matrices $A, B \in \mathbb{R}^{m \times n}$ be defined as

$$\langle A, B \rangle := \sum_{i=1}^{m} \sum_{j=1}^{n} a_{ij} b_{ij}.$$

We can generalize the above for higher dimensional arrays. Define the inner product between two arrays $A, B \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_m}$ to be

$$\langle A, B \rangle := \sum_{i_1=1}^{n_1} \cdots \sum_{i_m=1}^{n_m} a_{i_1 \cdots i_m} b_{i_1 \cdots i_m}.$$

Therefore we can now formulate the MAP as:

$$\begin{array}{lll} \mathrm{MAP}_{\circ}(m): & \min & \langle C, X \rangle \\ & \text{s.t.} & X \in \mathbf{X}^m, \end{array} \tag{11}$$

where C is the m-dimensional cost array, and $\mathbf{X}^m$ the set of all m-dimensional permutation arrays.

3 Matroid Intersection

The multidimensional assignment problem (MAP) can be formulated as the intersection of partition matroids as it will be shown in this section. A whole section is devoted to the subject, since this formulation is more general than all the previously mentioned formulations, and it presents a new way of looking into the problem which introduces many interesting open questions regarding the problem.

3.1 Definitions

Let us first state some necessary definitions from matroid theory [14].

Definition 3.1 *Given a finite set E and some family of subsets $\mathcal{F} \subset 2^E$, the set system $\mathcal{M} = (E, \mathcal{F})$ is called a* **matroid** *if the following axioms hold:*

(I1.) $\emptyset \in \mathcal{F}$.

(I2.) If $X \in \mathcal{F}$ *and* $Y \subset X \Rightarrow Y \in \mathcal{F}$.

(I3.) If $X, Y \in \mathcal{F}$ *and* $|X| = |Y| + 1 \Rightarrow \exists\, x \in X/Y$ *such that* $Y \cup \{x\} \in \mathcal{F}$.

If only (I1) and (I2) are true then the system is called **independent** *system.*

Axiom (I2) will be called the *subinclussiveness* property, while axiom (I3) the *accessibility* property. The elements of $\mathcal{F}$ are called **independent**, while those not in $\mathcal{F}$ are called **dependent**. Maximal independent sets are called **bases** and minimal dependent sets are called **circuits**. Given some independent system $(E, \mathcal{F})$, the **rank** of some $I \subseteq E$ is defined as

$$r(I) := \max\{|X| : X \subseteq I, X \in \mathcal{F}\},$$

while the **lower rank** as

$$lr(I) := \min\{|X| : X \subseteq I, X \in \mathcal{F}, X \cup \{s\} \notin \mathcal{F} \ \forall s \in I/X\}.$$

We can interpret the values of $r(E)$ and $lr(E)$ as the cardinality of the maximum cardinality basis and the minimum cardinality basis of the independent system respectively. Note that in the case where the independent system is a matroid, then $r(E) = lr(E)$. For any matroid $M = (E, \mathcal{F})$ there is an associated independent system on the same ground set $M^* = (E, \mathcal{F}^*)$ called the **dual** of M, such that the bases of M^* are the complements of the bases of M. For independent systems in general, the dual of an independence system $(E, \mathcal{F})$ is defined as$(E, \mathcal{F}^*)$ where

$$\mathcal{F}^* := \{I \subseteq E : \text{there is a basis } B \text{ of } (E, \mathcal{F}) \text{ such that } I \cap B = \emptyset\}$$

3.2 Formulation of the MAP(m)

In order to state the matroid intersection formulation, we will construct a hypergraph associated with the problem. Given $n_1, n_2, \ldots, n_m$ as above construct the m-partite hypergraph $H(V, E)$ as follows:

- $V := \bigcup_{i=1}^{m} V_i$, where $|V_i| = n_i$ and $V_i \cap V_j = \emptyset$ for all i, j.
- $E := \{(v_{i_1}, v_{i_2}, \ldots, v_{i_m}) : v_{i_j} \in V_j, j = 1, \ldots, m\}$, or in other words, E consists of only hyperedges on V of "size" m only. Equivalently we can define $E := V_1 \times \cdots \times V_m$.
- A cost function $c : E \to \mathbb{R}$, which is defined by the cost array C. That is, given any hyperedge $e = (v_{i_1}, v_{i_2}, \ldots, v_{i_m})$ the associated cost is $c(e) = C_{i_1 \cdots i_m}$.

Given any two hyperedges $e = (v_{i_1}, v_{i_2}, \ldots, v_{i_m})$ and $w = (v_{j_1}, v_{j_2}, \ldots, v_{j_m})$ we say that they are **disjoint** iff $v_{i_l} \neq v_{j_l}, l = 1, \ldots, m$. *In the MAP(m) we have to find* n_1 *disjoint hyperedges of minimum total weight.* In other words we have to find a *maximal* subset $I \subset E$ of disjoint hyperedges, with minimum weight $c(I) := \sum_{e \in I} c(e)$. If the problem is dense, which implies that the multidimensional array C is dense, then the cardinality of the maximal subsets I would be equal to n_1.

Now consider the following partitions of E as defined by $V_1, \dots, V_m$. Let

$$\Pi_{V_j} = \bigcup_{v \in V_j} P_{(V_j,v)}, \quad j = 1, \dots, m$$

be m partitions of E, where for each $v \in V_j$ we have the following set of hyperedges defined by it:

$$P_{(V_j,v)} := \{e \in E : e \text{ is incident to vertex } v \in V_j\}.$$

When we say that a hyperedge $e = (v_{i_1}, \dots, v_{i_m})$ is **incident** to some $v \in V_j$ we mean that $v_{i_j} = v$. If we define the *independence relationship* as the following family of subsets:

$$\mathcal{F}_{V_j} := \{I \subset E : |I \cap P_{(V_j,v)}| \leq 1, \text{for all } v \in V_j\}, \quad j = 1, \dots, m.$$

In words, $\mathcal{F}_{V_j}$ is a family of subsets of the hyperedges of H such that no subset contains any two hyperedges which are incident to the same vertex among those of V_j.

Theorem 3.1 *Given an MAP(m), the set system $M_{V_j} = (E, \mathcal{F}_{V_j})$ is a matroid, $j = 1, \dots, m$.*

Proof 3.1 *Axioms (I1) and (I2) are evidently true. For (I3), assuming that we have $X, Y \in \mathcal{F}_{V_j}$ with $|X| = |Y| + 1$, we have to show that $\exists e \in X/Y$ such that $Y \cup \{e\} \in \mathcal{F}_{V_j}$. Note that the edges of any given set $I \subset E$, due to the independence relationship, determine $|I|$ distinct sets $P_{(V_j,v)}$. If any two were not distinct that would imply that there exists some vertex $v \in V_j$ such that $|I \cap P_{(V_j,v)}| = 2$. Therefore since $|X| = |Y| + 1$, there exists some vertex $v \in V_j$ such that $|X \cap P_{(V_j,v)}| = 1$ and $|Y \cap P_{(V_j,v)}| = 0$, or in other words there exists an edge $e \in X/Y$ such that $Y \cup \{e\} \in \mathcal{F}_{V_j}$.* $\square$

In particular the independence system above is a **partition matroid**. In the MAP(m) we want to find some $I \in \bigcap_{i=1}^{m} \mathcal{F}_{V_j}$ of maximum cardinality and minimum weight $c(I)$. This is a weighted matroid intersection problem which will we define as follows:

MAP$_{\mathcal{M}}(m)$: *Given m matroids $M_{V_j} = (E, \mathcal{F}_{V_j})$ by their* **independent oracles**, *and a cost function $c : E \to \mathbb{R}$, find a set $I \in \bigcap_{i=j}^{m} \mathcal{F}_{V_j}$ of* **maximum cardinality** *such that its weight $\sum_{e \in I} c(e)$ its minimum.*

Note that the set system $(E, \bigcap_{j=1}^{m} \mathcal{F}_{V_j})$ is not a matroid, the intersection of matroids is not matroid in general (it is easy to give a counterexample for the specific system that shows that its not a matroid). We do have however

Corollary 3.1 *$(E, \bigcap_{j=1}^{m} \mathcal{F}_{V_j})$ is a independence system.*

It follows directly using theorem 3.1, since the intersection of matroids is an independence system [14].

```
algorithm independent($I$)
1   $\forall\, e = (v_{i_1}, v_{i_2}, \ldots, v_{i_m}) \in I$ do $\rightarrow$
2       do $j = 1, 2, \ldots, m \rightarrow$
3           if $v_{i_j}$ is labeled $\rightarrow$ STOP ($I \notin \mathcal{F}_{V_j}$)
4           else label $v_{i_j}$;
5       od;
6   od;
7   return $I \in \mathcal{F}_{V_j}$;
end independent;
```

Figure 3: Independence Oracle

3.3 Oracles

A word has to be said here regarding the *independence oracles* mentioned in the problem definition. An independent oracle can be considered as an algorithm that takes as an input a set $I \subseteq$ and answers the question whether this set is independent or not (i.e. that is satisfies the independence relationship). In our case it is trivial to see that given a set of hyperedges $I \subset E$ and some $j \in \{1, \ldots, m\}$, we can check whether $I \in \mathcal{F}_{V_j}$ in $\mathcal{O}(|I|)$ time by examining whether there are any two edges in I with their j^{th} component the same. Similarly we can check in linear time whether $I \in \bigcap_{i=1}^{m} \mathcal{F}_{V_j}$. The following simple labeling algorithm shown in Figure 3 can be considered as an independent oracle.

It is clear that the algorithm will require $\mathcal{O}(k|I|)$ computational time. Checking independence of a set is needed when the algorithm used for constructing the solution proceeds in an additive fashion, starting from the empty set and adding one element at a time until the set is independent. If we wish to proceed in the other way, that is starting from the ground set E and subtracting elements from it until we reach a solution, we need a different type of oracle, a *basis* oracle. A basis oracle can be considered as an algorithm which takes as an input a set $I \subseteq E$ and gives an answer to the question of whether or not this set contains a basis. The construction of such a basis oracle algorithm remains an open question.

3.4 Bounds

Establishing the fact that the system $(E, \bigcap_{j=1}^{m} \mathcal{F}_{V_j})$ is independent enables us to use existing results on bounds for **greedy** type algorithms for these type of problems. Specifically, given any $(E, \mathcal{F})$ be an independence system and some cost function $c : E \rightarrow \mathbb{R}_+$, a greedy type algorithm will produce a solution to the minimization problem by starting with the the ground set E and start excluding elements, always taking out the element with the largest weight. The algorithm will stop when the subset at hand is a basis, a fact which can be checked using the basis oracle mentioned in the previous section. The following result is due to [4].

Theorem 3.2 (Korte and Monma [1979]) *Let $(E, \mathcal{F})$ be an independence system. For $c : E \rightarrow \mathbb{R}_+$ let $G(E, \mathcal{F}, c)$ denote a solution found by a greedy type algorithm, while $OPT(E, \mathcal{F}, c)$ is the optimum solution. Then*

$$1 \leq \frac{G(E, \mathcal{F}, c)}{OPT(E, \mathcal{F}, c)} \leq \max_{F \subseteq E} \frac{|F| - lr^*(F)}{|F| - r^*(F)} = \rho(E, \mathcal{F})$$

for all $c : E \rightarrow \mathbb{R}_+$, where $r^*(F)$ and $lr^*(F)$ are the lower rank and rank respectively as defined on the dual $(E, \mathcal{F}^*)$. There is a cost function where the upper bound is attained.

Based on the above theorem, we can see that if we can evaluate the value of $\rho(E, \mathcal{F})$ we can establish a theoretical bound on the performance of a greedy type algorithm. Note however that the above result concerns all independent systems, and it is an open question whether or not for the independence system established for the multidimensional assignment problem, there is a closed form expression for $\rho(E, \mathcal{F})$, which in turn would imply an ϵ-approximation algorithm for the problem. It is conjectured that we have $\rho(E, \bigcap_{j=1}^{m} \mathcal{F}_{V_j}) = f(m)$, that is the value of ρ is some constant function on the dimension of the problem m.

4 Data Structures

It immediately follows from the nature of the problem that an efficient way of handling the large data is required. There are two basic problems in handling the input data of an instance of the MAP. First, multidimensional arrays are not desirable in terms of memory requirements, and second, the variable dimension m poses implementation problems. Both of these problems are solved if we transform the multidimensional array into a one-dimensional array.

4.1 Transformation into one dimension

Let $\pi_l := \prod_{i=1}^{l} n_i$ for $l = 1, \ldots, m$. The array $C \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_m}$ is stored as an one-dimensional array $c \in \mathbb{R}^{\pi}$, in **column** format. For example if $C \in \mathbb{R}^{2\times2\times2}$ then the array $c \in \mathbb{R}^8$ will be

$$c = (C_{111}, C_{211}, C_{121}, C_{221}, C_{112}, C_{212}, C_{122}, C_{222}).$$

In addition to the sets of integers N_i, $i = 1, \ldots, m$ defined previously define also the set $N := \{1, 2, \ldots, \pi\}$. The set $N_1 \times \cdots \times N_m$ can be regarded as the index set of C while N the index set of c. The transformation between an index $(i_1, i_2, \ldots, i_m)$ of C to an index k of c is given by the following function $f : N_1 \times \cdots \times N_m \rightarrow N$

$$f(i_1, i_2, \ldots, i_m) := i_1 + \sum_{j=2}^{m} \pi_{j-1}(i_j - 1) = k. \tag{12}$$

The inverse (from an index k of c to an index $(i_1, i_2, \ldots, i_m)$ of C) can be obtained in a similar fashion from the function $f^{-1} : N \rightarrow N_1 \times \cdots \times N_m$, the inverse of f, which is defined by

$$f^{-1}(k) := \begin{pmatrix} \left\lfloor k - 1 - \sum_{l=1}^{m-1} \pi_{m-l}(i_{m-l+1} - 1) \right\rfloor + 1 \\ \vdots \\ \left\lfloor \frac{k-1-\sum_{l=1}^{m-q} \pi_{m-l}(i_{m-l+1}-1)}{\pi_{m-1}} \right\rfloor + 1 \\ \vdots \\ \left\lfloor \frac{k-1}{\pi_{m-1}} \right\rfloor + 1 \end{pmatrix}^T = \begin{pmatrix} i_1 \\ \vdots \\ i_q \\ \vdots \\ i_m \end{pmatrix}^T .$$

Lemma 4.1 *The function f is a bijection.*

Proof 4.1 *Since $|N_1 \times \cdots \times N_m| = |N| = \pi$ by construction, for f to be bijective it suffices to show that for any two $(i_1, \ldots, i_m), (j_1, \ldots, j_m) \in N_1 \times \cdots \times N_m$, $(i_1, \ldots, i_m) \neq (j_1, \ldots, j_m)$ (i.e. distinct elements), we have $f(i_1, \ldots, i_m) \neq f(j_1, \ldots, j_m)$. Clearly this is true for $m = 1$, since $f(i_1) = i_1 \neq j_1 = f(j_1)$. Suppose that is true for m and let $(i_1, \ldots, i_m + 1) \neq (j_1, \ldots, j_m + 1)$. Then we have*

$$f(i_1, \ldots, i_m + 1) = f(i_1, \ldots, i_m) + \pi_m(i_{m+1} - 1),$$

$$f(j_1, \ldots, j_m + 1) = f(j_1, \ldots, j_m) + \pi_m(j_{m+1} - 1).$$

There are three cases: i) $(i_1, \ldots, i_m) \neq (j_1, \ldots, j_m)$ and $i_{m+1} = j_{m+1}$, ii) $(i_1, \ldots, i_m) = (j_1, \ldots, j_m)$ and therefore $i_{m+1} \neq j_{m+1}$, and
iii) $(i_1, \ldots, i_m) \neq (j_1, \ldots, j_m)$ and $i_{m+1} \neq j_{m+1}$. In all of the cases $f(i_1, \ldots, i_m + 1) \neq f(j_1, \ldots, j_m + 1)$ since we have respectively for the first case the induction hypothesis and for the second case $i_{m+1} \neq j_{m+1}$. For the third case, note that $f(i_1, \ldots, i_m), f(j_1, \ldots, j_m) \in [1, \ldots, \pi_m]$ and $\pi_m(i_{m+1} - 1), \pi_m(j_{m+1} - 1) \in [0, \pi_m, \ldots, (n_{m+1} - 1)\pi_m]$, which imply the following inequalities considering that we have distinct elements

$$1 \leq \ |f(i_1, \ldots, i_m) - f(j_1, \ldots, j_m)| \ \leq \pi_m - 1,$$

$$|\pi_m(i_{m+1} - 1) - \pi_m(j_{m+1} - 1)| \ \geq \pi_m.$$

Therefore, we have that

$$\pi_m(i_{m+1} - 1) \neq \pi_m(j_{m+1} - 1) + (f(j_1, \ldots, j_m) - f(i_1, \ldots, i_m)),$$

which implies that $f(i_1, \ldots, i_m + 1) \neq f(j_1, \ldots, j_m + 1)$. $\square$

From f and f^{-1} it is clear that we can construct the c array as $c(i) = C(f^{-1}(i)), i = 1, \ldots, \pi$.

4.2 The index structure of c

Assume that we are given an element $(i_1, i_2, \ldots, i_m)$ of C, and for a given k we wish to generate the set

$$N \supset S_{i_k} := \{i \in N \ : \ f^{-1}(i) = (i_1, \ldots, i_k, \ldots, i_m)\},$$

that is the set of all elements in N such that their corresponding element in $N_1 \times \cdots \times N_m$ contains i_k. A direct approach would be to generate the set $S'_{i_k} \subset N_1 \times \cdots \times N_m$ defined as

$$S'_{i_k} := \{(i_1, \ldots, i_k, \ldots, i_m) \ : \ i_j = 1, \ldots, n_j, j = 1, \ldots, m, j \neq k\},$$

and then take the image of S'_{i_k} under f, $S_{i_k} = f(S'_{i_k})$, a process which will require $\mathcal{O}(m\frac{\pi}{n_k})$ time. The way that c is constructed, it follows that its **index structure**, that is the relationship between the indices of c and those of C, is well defined. Let us define for expository purposes, an ordered π-tuple which corresponds to the index set N of c, as $I := (1, 2, \ldots, \pi - 1, \pi)$. For every $i \in I$ there is a corresponding element $(i_1, i_2, \ldots, i_m)$ in the index set of C. Define the following for every dimension $k = 1, 2, \ldots, m$ and element $i_k \in N_k$:

$$\sigma_{i_k} := (i_k - 1)\pi_{k-1} + 1,$$

$$\nu_k := \frac{\pi}{\pi_k}.$$

According to the above values, for any i_k we can identify the following π_{k-1}-sub-tuples contained in I,

$$I = (1, 2, \ldots, \underbrace{B_1^{i_k}, \ldots,}_{\pi_k} \underbrace{B_2^{i_k}, \ldots,}_{\pi_k} \underbrace{B_3^{i_k}, \ldots,}_{(\nu_k - 3)\pi_k} B_{\nu_k}^{i_k}, \ldots, \pi - 1, \pi),$$

where $B_l^{i_k} = ((l-1)\pi_k + \sigma_{i_k}, (l-1)\pi_k + \sigma_{i_k} + 1, \ldots, (l-1)\pi_k + \sigma_{i_k} + \pi_{k-1} - 1)$ for $l = 1, \ldots, \nu_k$. So for every i_k, I contains ν_k sub-tuples of size π_{k-1}, starting from the element σ_{i_k} and occurring every π_k elements.

Lemma 4.2 $S_{i_k} = \bigcup_{l=1}^{\nu_k} B_l^{i_k}$.

Proof 4.2 *Since $S_{i_k} = f(S'_{i_k})$ it suffices to show that $f(S'_{i_k}) = \bigcup_{l=1}^{\nu_k} B_l^{i_k}$ where, considering the fact that f is a bijection, it would be true if for any $(i_1, \ldots, i_m) \in S'_{i_k}$ we have $f(i_1, \ldots, i_m) \in \bigcup_{l=1}^{\nu_k} B_l^{i_k}$. For f defined as in (12) and any $(i_1, \ldots, i_k, \ldots, i_m) \in S'_{i_k}$ we have*

$$\begin{aligned} f(i_1, \ldots, i_k, \ldots, i_m) &= i_1 + \sum_{\substack{j=2 \\ j \neq k}}^{m} \pi_{j-1}(i_j - 1) + \pi_{k-1}(i_k - 1) \\ &= f(i_1, \ldots, i_{k-1}) + \pi_{k-1}(i_k - 1) + \sum_{j=k+1}^{m} \pi_{j-1}(i_j - 1). \end{aligned}$$

The ranges for $f(i_1, \ldots, i_{k-1})$ and $\sum_{j=k+1}^{m} \pi_{j-1}(i_j - 1)$, are $[1, 2, \ldots, \pi_{k-1}]$ and $[0, \pi_k, 2\pi_k, \ldots, \pi_{k+1} - \pi_k, \pi_{k+1}, \ldots, \pi - \pi_k]$ respectively, while the term $\pi_{k-1}(i_k - 1)$ is constant. Combining these ranges we can see that $f(i_1, \ldots, i_k, \ldots, i_m) \in [B_1^{i_k}, B_2^{i_k}, \ldots, B_{\nu_k}^{i_k}]$. □

It follows from lemma 4.2 that we can generate $\bigcup_{l=1}^{\nu_k} B_l^{i_k}$ and therefore S_{i_k} in $\mathcal{O}(\frac{\pi}{n_k})$ time.

5 An Algorithm for the MAP

The algorithm which we will use for solving the MAP(m) follows the general methodology of Greedy Randomized Adaptive Search Procedures (GRASP). GRASP is an iterative procedure consisting of two phases at each iteration, a construction phase and a local search phase. In the first phase, a solution is constructed in a greedy randomized fashion, while in the second phase, a local search is applied in the neighborhood of the constructed solution, for possible further improvement. This process is repeated until a stopping criterion, such as maximum number of iterations, is satisfied. Each iteration can be viewed as a procedure which samples high-quality points from the solution space, and applying a local search to the point chosen to obtain a local optimum For a tutorial and survey of GRASP, see Feo and Resende [2]. Figure 4 shows the pseudo-code for a generic GRASP.

GRASP has been implemented to solve Quadratic Assignment Problems (QAP)s with dense [7, 5] and sparse coefficient matrices [11], while an implementation for solving the BiQuadratic Assignment Problem can be found in [8]. A parallel implementation of a GRASP for solving the QAP is described in [10]. In the sections that follow, the construction phase and the local search phase of the algorithm as applied to the MAP(m) are described. It is noted that the formulation used for the development of the algorithm, is the one presented on §2.2 using permutations.

```
procedure GRASP(ListSize,MaxIter,RandomSeed)
    InputInstance();
    do $k = 1, \dots,$ MaxIter $\rightarrow$
        ConstructSolution(ListSize,RandomSeed);
        LocalSearch(BestSolutionFound);
        UpdateSolution(BestSolutionFound);
    od;
    return BestSolutionFound
end GRASP;
```

Figure 4: Pseudo-code for a generic GRASP

5.1 Construction of a Solution

The objective function of the MAP is nothing else but the summation of n_1 elements of C such that the permutation constraints are satisfied. In the construction phase we will construct a solution in a greedy randomized manner. Let us define the set of already made assignments upon making the k^{th} assignment as

$$\Gamma_k := \{i \in N \;:\; f^{-1}(i) = (p_1(j), \dots, p_M(j)) \text{ for some } j = 1, \dots, k-1\},$$

and the set of infeasible assignments resulting from assigning s in the solution

$$\Delta_s := \{i \in N \;:\; \text{for } f^{-1}(s) = (i_1, \dots, i_M), i \in \bigcup_{j=1}^{M} S_{i_j}\}.$$

Our RCL will be represented by a combination of a doubly linked list, with a pointer array. The array `key()` is the c array sorted in increasing order, and the `inv()` array is just a pointer array that shows the position in the `key()` of any index of c. The arrays `prev()` and `next()` are used for the doubly linked list, that is to *move* in the `key()` array.

Figure 5: Doubly linked lists

Example 5.1 *Consider for example Figure 5 where the c array and the* `inv()`, `prev()`, `key()` *and* `next()` *arrays are as shown. Assume that we make the first assignment, and choose randomly among the first* $\lfloor \alpha\pi \rfloor$ *smallest elements of* `key()`*, and assume that we pick*

```
procedure ConstructionPhase($\alpha$)
1     $T = \pi$ : $\Gamma_0 = \emptyset$;
2     do $k = 1, \ldots, n_M$ $\rightarrow$
3          $t$ =random[1, $\lfloor \alpha T \rfloor$];
4          $s$ =startkey;
5          do $i = 2, \ldots, t$ $\rightarrow$
6               $s$ =next($s$);
7          od;
8          $s$ =key($s$);
9          $\Gamma_k = \Gamma_{k-1} \cup \{s\}$;
10         GenerateDelta($s, \Delta_s$);
11         for $i \in \Delta_s$ $\rightarrow$
12              $T = T - 1$;
13              UpdateLists($i$,prev,next,key);
14              $\Delta_s = \Delta_s / \{i\}$;
15         rof;
16    od;
17    return $\Gamma_{n_M}$
end ConstructionPhase;
```

Figure 6: Pseudo-code for GRASP construction phase

the element `key(2)`*=2. Then* $\Gamma_1 = \{2\}$ *and assume that* $\Delta_2 = \{2, 5\}$*, where* Δ_2 *is generated as described previously. We update our RCL as follows:*
For element 2,
`key(inv(2))=-1,`
`prev(next(inv(2)))=prev(inv(2)),`
`next(prev(inv(2)))=next(inv(2)),`
and for element 5,
`key(inv(5))=-1,`
`prev(next(inv(5)))=prev(inv(5)),`
`next(prev(inv(5)))=next(inv(5)).`
Every time we choose an element from the `key()` *array we proceed using the* `prev(), next()` *arrays.*

The construction phase is shown in pseudo-code in Figure 6. In line 1 we initialize the size or our RCL, and the set Γ. In the loop defined by the lines 2-16, all the assignments are made. After generating a random number in line 3, in lines 4-8 we *move* in the doubly linked list to retrieve the index s which corresponds to our current assignment. We update Γ in line 9, and in line 10 we generate the set of infeasible assignments due to s, the set Δ_s. The procedure `GenerateDelta`(s, Δ_s) is done as described in 4.2. In lines 11-15 we update the RCL, by excluding from the doubly linked list the infeasible elements contained in Δ_s. The procedure `UpdateLists`(i,`prev`,`next`,`key`) is described earlier.

5.2 Local Search

After completion of the construction phase, we have a feasible solution which is consisted of m n_1-permutations. In the local search phase, we try to improve each solution constructed in phase 1 by searching its neighborhood for a better solution.

Let p be a r-permutation of n elements ($r \leq n$). Define the *difference* between p and a distinct r-permutation q, to be

$$\delta_r(p,q) := \{i \ : \ p(i) \neq q(i), i = 1,\dots,r\}. \tag{13}$$

Define the *distance* between them as

$$d_r(p,q) := |\delta_r(p,q)|. \tag{14}$$

The *k-exchange neighborhood* of p is the set of all permutations q such that $d_r(p,q) \leq k$,

$$N_k(p) := \{q \ : \ d_r(p,q) \leq k, \ 1 \leq k \leq r\}. \tag{15}$$

Remark 5.1 $|N_2(p)| = \binom{r}{2} + r(n-r)$.

For a given $P \in \mathbf{P}^m$ define its 2-exchange neighborhood as the set

$$N_2(P) := \{Q \in \mathbf{P}^m \ : \ d_m(p_j,q_j) \leq 2 \text{ for all } j = 1,\dots,m\}. \tag{16}$$

Remark 5.2 $|N_2(P)| = \prod_{j=1}^{m}\left(\binom{n_m}{2} + n_m(n_j - n_m)\right)$.

A local search algorithm based on this neighborhood definition would run in exponential time, therefore we have to employ a different neighborhood definition which is smaller in size. Let

$$N_2(P) \supset \hat{N}_2(P) := \{Q \in \mathbf{P}^m \ : \ d_m(p_j,q_j) \leq 2 \text{ for some } j = 1,\dots,m\}. \tag{17}$$

Therefore each $Q \in \hat{N}_2(P)$ differs from P in only one column.

Remark 5.3 $|\hat{N}_2(P)| = m\binom{n_m}{2} + n_m \sum_{j=1}^{m}(n_j - n_m)$.

For every $Q \in \hat{N}_2(P)$ we can associate a triplet (j,k,l) where $j = 1,\dots,m$, $k = 1,\dots,n_m$ and $l = k,\dots,n_j$, which indicates that $p_j(k) = q_j(l)$ and $p_j(l) = q_j(k)$. If $l \leq n_m$ then $d_m(p_j,q_j) = 2$ while if $n_m < l \leq n_j$ then $d_m(p_j,q_j) = 1$.

Next we define the gain (either positive or negative) between two solutions in a neighborhood. Let

$$Z(P) = \sum_{i=1}^{n_1} C_{ip_2(i)\cdots p_{m-1}(i)p_m(i)} \tag{18}$$

for some $P \in \mathbf{P}^m$. Note that $Z(P)$ is equivalent with the objective function defined in (10). Then it is immediate that for any $Q \in \hat{N}_2(P)$

$$Z(P) - Z(Q) = \sum_{i \in \delta(p_j,q_j)} C_{p_1(i)\cdots p_m(i)} - C_{q_1(i)\cdots q_m(i)}, \tag{19}$$

and due to the definition of $\hat{N}_2(P)$ there is only one j such that $\delta(p_j,q_j) \neq \emptyset$. Having defined (17) and (19), a local search procedure for finding local optima is straightforward. Given any P we generate the gains $Z(P) - Z(Q)$ for every $Q \in \hat{N}_2(P)$, and we pick the Q with the largest positive gain. The procedure is repeated in $\hat{N}_2(Q)$ until no positive gains can be found, that is, our current solution is the locally optimum.

6 Problem Generator

The efficiency of an algorithm for a given combinatorial optimization problem that belongs to the NP-Hard complexity class, is determined by several criteria including the accuracy of the solution, the speed of the algorithm, and the effectiveness of the algorithm with respect to different problem instances. Since it is unlikely (unless P=NP) that a polynomial time algorithm could be devised that would solve the problem efficiently, we can only rely on heuristic procedures that will provide good solutions in reasonable computational time.

In evaluating the performance of heuristic algorithms it is crucial to have a set of good benchmark problem instances, to perform the computational experiments. Basically there are two types of benchmark problems for a given problem, the first involves instances which come from some real-world application, and the second are instances which are the outcome of some **problem generator** that was designed to construct random instances for the particular problem. While instances from a real-world application are desirable, nevertheless they have the dissadvantage that they tend to present trends in the data related to the specific application, thereby an algorithm which performs good at a particular benchmark problem set might not necessarily perform the same for some other set coming from a different application. Moreover, the availability of instances from real-world applications is rather limited. Therefore, the design of good problem generators for a particular problem is desirable. Various characteristics such as instance size, hardness, sparsity etc. could also be implemented in instances generated as such.

6.1 A Random Generator

Generally it is hard to construct a good problem generator for a combinatorial optimization problem, and not so much research has been devoted to the subject [12, 3, 1, 6]. One may conjecture that the problem of *finding an algorithm that will generate problem instances that are NP-hard for a given problem class* is itself an NP-hard problem. The characteristics that a good problem generator should have can be the following:

1. problem instance characteristics such as size, sparsity etc. should be a parameter of the generator algorithm.
2. the problem instances that it generates should not be *easy* to solve by any heuristic or exact method.
3. the *hardness* of the instances generated should be a parameter of the generator algorithm.

The first item on the list above is easy to implement. However items 2 and 3 are not trivial.

Here we propose an initial problem generator that generates random instances of the MAP. The algorithm is simple, and it proceeds by first generating a random multidimensional array C, where the costs are random numbers distributed uniformly in the interval $[2, M]$ where M is a parameter of the generator. Then all the *diagonal* elements of the array are set to 1, thereby making the optimal solution to be n_1 (i.e. the smallest range), and the optimal permutations are the identity permutations.

7 Computational Results

In this section, computational results that illustrate the performance of the algorithm for solving the multidimensional assignment problem are presented. The algorithm implemented follows the description given in §5, while the test problems were generated using

```
algorithm ConstructRandomInstance($T, m, n_1, n_2, \ldots, n_m$)
1   $\pi := \prod_{i=1}^{m} n_i$
2   do $i = 1, 2, \ldots, \pi \rightarrow$
3       $(i_1, i_2, \ldots, i_m) = f^{-1}(i)$;
4       $C_{(i_1, i_2, \ldots, i_m)} =$ random$[2, T]$;
5   od;
6   do $i = 1, 2, \ldots, n_1 \rightarrow$
7       $C_{(i, i, \ldots, i)} = 1$;
8   od;
9   return $C$;
end ConstructSparse;
```

Figure 7: MAP random instance generator

the problem generator described in §6. Overall twenty one problems were generated of various sizes and dimension. For each problem the GRASP algorithm was allowed to run for a maximum of 100000 iterations or until the optimum solution was found. The computational runs were performed on a Intel Pentium II 400 MHz computer, with 128MBytes of memory, running the Linux operating system. For each problem instance the number of iterations and the CPU time until the algorithm found the optimum solution was recorded. If the optimum was not found, the aforementioned values correspond to the best solution found by the algorithm.

From the computational results in Table 1 we can see that the algorithm finds the optimum solution in all but three instances. The instances for which the optimum is not found within the maximum number of iterations, are amongst the largest ones in terms of number of elements.

8 Conclusions and Future Research

The primary subject of study in this research is the multidimensional assignment problem (MAP) as it arises from its application in a multiple-sensor multiple-target tracking framework. The MAP is an NP-Hard combinatorial optimization problem, which has been studied before in the past. It is a generalized version of the well known linear assignment problem, and it seems to enjoy various applications from different fields.

In this research we provide new mathematical formulations of the problem, all of them equivalent to each other, and to the classical integer programming formulation. Using the permutation formulation of the problem, which has an inherent combinatorial nature, a purely discrete heuristic algorithm for providing close to optima solutions to the problem is presented. In order for the algorithm to perform satisfactorly fast, efficient data structures to handle the massive data associated with the problem are developed, and integrated to the algorithm. The result is a heuristic algorithm that can solve MAP's of varying dimension. Moreover, using the same permutation formulation of the problem, a problem generator that generates MAP instances of known optimal solution is developed and implemented. The instances generated are purely for testing purposes. All algorithms have been implemented in FORTRAN, and copies of working codes are provided.

An interesting aspect that emerged in the process of this research, is the formulation of the problem as a matroid intersection problem. Using results from the rich matroid theory, this new approach might lead to new and powerfull solution methods for the problem. Specifically the following open questions emerged:

name	dimension	size	optimum	GRASP Solution iterations	CPU time	solution
6x4	6	4096	4	364	0m1s	4
4x10	4	10000	10	166	0m2s	10
3x22	3	10648	22	129	0m05s	22
6x5	6	15625	5	2228	0m35s	5
5x7	5	16807	7	7234	2m15s	7
7x4	7	16384	4	2204	0m36s	4
5x8	5	32768	8	1805	1m23s	8
4x14	4	38416	14	288	0m19s	14
6x6	6	46656	6	2627	3m16s	6
4x15	4	50625	15	5078	7m51s	15
3x38	3	54872	38	176	0m53s	38
5x9	5	59049	9	9228	16m04s	9
3x40	3	64000	40	149	0m37s	40
8x4	8	65536	4	5470	11m30s	4
7x5	7	78125	5	2228	5m37s	5
3x46	3	97336	46	1772	15m55s	46
5x10	5	100000	10	1477	4m33s	10
6x7	6	117649	7	7651	30m40s	15*
4x20	4	160000	20	4876	26m10s	20
7x6	7	279936	6	3226	10m22s	12*
8x5	8	390625	5	662	15m10s	13*

Table 1: Summary of computational runs (* indicates not optimal)

- How does the dual problem structure looks like, and if there is a possibility to develop a primal-dual algorithm for solving the problem. Such primal-dual algorithms on matroid intersections that involve two matroids are optimal, but in the case of MAP where we have the intersection of more than two matroids, an approximation algorithm could be developed.
- Is there an efficient way to compute or approximate (bound) the value of $\rho(E, \mathcal{F})$? If such an algorithm is constructed, then we immediately have an approximation algorithm for the minimazation version of the MAP(m). Note that for the maximization version of the problem, a similar approach gives an ϵ-approximation algorithm with $\epsilon = 1/m$.
- Develop a basis oracle for the independent system $(E, \bigcap_{j=1}^{m} \mathcal{F}_{V_j})$. That is, an algorithm that given some $I \subseteq E$ will determine whether or not I contains a maximal independent set.
- Develop a problem generator using the matroid intersection formulation.

9 FORTRAN Codes

9.1 Problem Generator

```
c       ---------------------------------------------
c       Program to generate random input data for n-dimensional
c       assignment problem. The elements of the cost matrix C are
```

```
c     between 1-100. The problem instances have known optimal
c     solutions, the identity permutations, with optimal solution value
c     equal to Nmin := the minimum of N1,...,Ndim.
c
c     Author: Pitsoulis Leonidas
c     ----------------------------------------------
c     Inputs from user:
c
c     dim          -  the dimension
c     N1,...,Ndim -  the range for each index 1..dim
c
      integer nmax,dimmax,npower
      parameter (nmax=100,dimmax=10)
      parameter (npower=10000000)
      integer  c(npower), nindex(dimmax), indx(dimmax)
      integer dim, n
      integer   seed,high,i, j,k
      real    randp,xrand
c     ----------------------------------------------
      seed=2700001
      high=100

      print*,'Enter dimension dim:'
      read*, dim
      print*,'Enter N1,...,Ndim (in ascending order):'
      read*, (nindex(i), i=1,dim)
      n=1
      do 10 i=1,dim
         n=n*nindex(i)
 10   continue
      do 20 i=1,n
         xrand  = randp(seed)
         nselct = 1 + seed/(2147483647/high)
         c(i) = nselct + 1
 20   continue

      do 30 i=1,nindex(1)
         do 40 k=1,dim
            indx(k) = i
 40      continue
         call indxi(n,dim,indx,j,nindex)
         c(j) = 1
 30   continue

      do 50 i=1,n
         print *, c(i)
 50   continue


      end

      real function randp(ix)
c     ----------------------------------------------
c     randp: Portable pseudo-random number generator.
c            Reference: L. Schrage, "A More Portable Fortran
```

```
c           Random Number Generator", ACM Transactions on
c           Mathematical Software, Vol. 2, No. 2, (June, 1979).
c     ---------------------------------------------
      integer a,p,ix,b15,b16,xhi,xalo,leftlo,fhi,k
      data a/16807/,b15/32768/,b16/65536/,p/2147483647/
c     ---------------------------------------------
      xhi    = ix/b16
      xalo   = (ix - xhi*b16)*a
      leftlo = xalo/b16
      fhi    = xhi*a + leftlo
      k      = fhi/b15
      ix     = (((xalo - leftlo*b16) - p) + (fhi - k*b15)*b16) + k
      if (ix .lt. 0) ix = ix + p

      randp = float(ix)*4.656612875e-10
c     ---------------------------------------------
      return

c     end of randp
      end
```

9.2 Data Structures

9.2.1 The Cost Array Functions

```
     subroutine iindx(n,dim,i,nindex,indx)
c     -----------------------------------------------------------------
c     Given the dimension dim of the multi-dim array C, which is
c     represented as 1-dim array c(1)...c(n), then given the index i
c     of some element c(i) return the dim-indices (dim=2,3,..etc.) that
c     correspond, in the array indx().
c     -----------------------------------------------------------------
      integer nindex(dim),indx(dim)
      integer n,dim,i,j,l
      real p
      l=i
      p=real(n)
      do 10 j=dim,1,-1
         p=p/(real(nindex(j)))
         indx(j)=int((real(l-1))/p) + 1
         l = l - (indx(j) - 1 )*int(p)
10    continue
      return
      end

      subroutine indxi(n,dim,indx,j,nindex)
c     -----------------------------------------------------------------
c     Given the dim-indices (dim=2,3,...etc.) in the array indx(),
c     return the index j
c     -----------------------------------------------------------------
      integer nindex(dim),indx(dim)
      integer n,dim,j,k
      real p
      p=real(n)
      j=0
      do 10 k=dim,1,-1
         p=p/(real(nindex(k)))
         j=j+p*(indx(k)-1)
10    continue
      j=j+1
      return
      end
```

9.2.2 The Queues

```
      subroutine insrtq(n,v,iv,sizeq,q,iq)
c     ------------------------------------------------------------------
c     insrtq: Insert an element (v,iv) into a queue (q,iq).
c     ------------------------------------------------------------------
c     Passed scalars:
c
c     n      - size of c
c     v      - heap element (value)
c     iv     - heap element (index)
c     sizeq  - size of heap
c
c     ------------------------------------------------------------------
      integer n,sizeq,iv,v
c     ------------------------------------------------------------------
c     Passed arrays:
c
c     q      - heap (value)
c     iq     - heap (index)
c
c     ------------------------------------------------------------------
      integer iq(n),q(n)
c     ------------------------------------------------------------------
c     Local scalars:
c
c     sq     - temporary size of heap
c     tsz    - temporary variable (sq/2)
c     w      - temporary variable
c     iw     - temporary variable
c
c     ------------------------------------------------------------------
      integer sq,tsz,w,iw

c     ------------------------------------------------------------------
c     Insert element into heap.
c     ------------------------------------------------------------------
      sizeq=sizeq+1
      q(sizeq)=v
      iq(sizeq)=iv
c     ------------------------------------------------------------------
c     Update heap to proper order.
```

```
c     ------------------------------------------------------------------
      sq=sizeq
      w=q(sq)
      iw=iq(sq)
10    tsz=sq/2
      if (tsz.ne.0) then
         if (q(tsz).ge.v) then
            q(sq)=q(sq/2)
            iq(sq)=iq(sq/2)
            sq=sq/2
            goto 10
         endif
      endif
      q(sq)=w
      iq(sq)=iw
      return
      end



      subroutine removq(n,v,iv,sizeq,q,iq)
c     ------------------------------------------------------------------
c     removq: Remove smallest element (v,iv) from a priority
c             queue (q,iq).
c     ------------------------------------------------------------------
c     Passed scalars:
c
c     n      - size total
c     v      - smallest element in heap (value)
c     iv     - smallest element in heap (index)
c     sizeq  - size of heap
c
c     ------------------------------------------------------------------
      integer n,v,iv,sizeq
c     ------------------------------------------------------------------
c     Passed arrays:
c
c     q      - heap (value)
c     iq     - heap (index)
c
c     ------------------------------------------------------------------
      integer iq(n),q(n)
c     ------------------------------------------------------------------
c     Local scalars:
c
c     vtmp   - tmp smallest element in heap (value)
c     ivtmp  - tmp smallest element in heap (index)
c     k      - heap counter
c     j      - heap counter (2*k)
c     szqd2  - sizeq/2
c
c     ------------------------------------------------------------------
      integer vtmp,ivtmp,k,j,szqd2
c     ------------------------------------------------------------------
c     Remove element from heap.
```

```
c     ------------------------------------------------------------------
      v=q(1)
      iv=iq(1)
      q(1)=q(sizeq)
      iq(1)=iq(sizeq)
      sizeq=sizeq-1
c     ------------------------------------------------------------------
c     Update heap to proper order.
c     ------------------------------------------------------------------
      k=1
      vtmp=q(k)
      ivtmp=iq(k)
      szqd2=sizeq/2
10    if (k .gt. szqd2) goto 20
   j=k+k
   if (j .lt. sizeq) then
if (q(j) .gt. q(j+1)) j=j+1
   endif
   if (vtmp .le. q(j)) goto 20
   q(k)=q(j)
   iq(k)=iq(j)
   k=j
   goto 10
20    continue
      q(k)=vtmp
      iq(k)=ivtmp
      return
      end
```

9.3 Local Search

```
      subroutine local(n,dim,nmax,dimmax,nsmall,c,pm,nindex,objv,
     +                 indexi,indexj)
c     ------------------------------------------------------------------
c     Perform local search in the solution pm(.,.) with objv as the
c     objective function value.
c     ------------------------------------------------------------------
c     ------------------------------------------------------------------
c     Passed scalars and arrays:
c     ------------------------------------------------------------------
      integer dim,dimmax,nmax,nsmall,objv,n
      integer c(n),nindex(dim),pm(dimmax,nmax),indexi(dim),indexj(dim)
c     ------------------------------------------------------------------
c     Local scalars and arrays:
c     ------------------------------------------------------------------
      integer gain,diff,i,j,k,l,io,jo,in,jn,kk,temp,ic,jc,kc
      real p
10    gain=0
      do 20 k=1,dim
         do 30 i=1, nsmall
            do 40 j=i+1, nindex(k)
               do 50 l=1,dim
                  indexi(l)=pm(l,i)
                  indexj(l)=pm(l,j)
50             continue
```

```
               temp=indexi(k)
               indexi(k)=indexj(k)
               indexj(k)=temp
               p=real(n)
               io=0
               jo=0
               in=0
               jn=0
               do 60 kk=dim,1,-1
                  p=p/(real(nindex(kk)))
                  io=io+p*(pm(kk,i)-1)
                  jo=jo+p*(pm(kk,j)-1)
                  in=in+p*(indexi(kk)-1)
                  jn=jn+p*(indexj(kk)-1)
60             continue
               io=io+1
               jo=jo+1
               in=in+1
               jn=jn+1
               if (j.le.nsmall) then
                  diff= c(io)+c(jo)-c(in)-c(jn)
               else
                  diff= c(io)-c(in)
               endif
               if (diff.gt.gain) then
                  gain=diff
                  ic=i
                  jc=j
                  kc=k
               endif
40          continue
30       continue
20    continue
      if (gain.gt.0) then
         objv=objv-gain
         temp=pm(kc,ic)
         pm(kc,ic)=pm(kc,jc)
         pm(kc,jc)=temp
         goto 10
      endif
      return
      end
```

9.4 Construction of a Greedy Solution

```
      subroutine construct(n,dim,nmax,dimmax,nsmall,c,pm,inv,prev,next,
     +              key,seed,alpha,nindex,indx,batch,numbatch,period,
     +                  objv)
c     ------------------------------------------------------------------
c     Construct a greedy randomized feasible solution
c     ------------------------------------------------------------------
      integer n,dim,nmax,nsmall,i,j,k,l,m,objv,dimmax
      integer c(n),inv(n),prev(n),next(n),key(n)
      integer nindex(dim),indx(dim),pm(dimmax,nmax)
      integer batch(dim),numbatch(dim),period(dim)
      integer high,rclsiz,seed,high,nselct,start,chosen,index,exi,exj
      real*4  randp,xrand,alpha
      rclsiz=n
      start=1
      objv=0
      do 10 i=1,nsmall
c     ---------------------------------------------------------------
c     Select element, at random, from the best (rclsiz)*alpha cost
c     elements.
c     ---------------------------------------------------------------
         xrand=randp(seed)
         high=alpha*(rclsiz)
         if (high.lt.1) then
            high=1
         endif
         nselct=1+seed/(2147483647/high)
c     ---------------------------------------------------------------
c     The variable chosen will be the nselct-th smallest element chosen
c     from the double linked list, i.e. key(chosen)=index in c() of
c     element being chosen.
c     ---------------------------------------------------------------
         chosen=start
         do 20 j=2,nselct
               chosen=next(chosen)
20       continue
         index=key(chosen)
c     ---------------------------------------------------------------
c     Exclude from the doubly-linked list all the elements that are not
c     feasible anymore due to the insertion of the c(index) element
c     in the constructed solution.
c     ---------------------------------------------------------------
         call iindx(n,dim,index,nindex,indx)
         do 25 l=1, dim
            do 26 m=i,nindex(l)
               if (pm(l,m).ne.indx(l)) then
                  goto 26
               endif
               temp=pm(l,i)
               pm(l,i)=pm(l,m)
               pm(l,m)=temp
               goto 25
26          continue
25       continue
         objv=objv+c(index)
           do 30 j=1,dim
            exi=(indx(j)-1)*batch(j)
            do 40 k=1, numbatch(j)
               do 50 m=1, batch(j)
                  exi=exi+1
c     ---------------------------------------------------------------
c     exclude the element exi from the doubly linked list
c     ---------------------------------------------------------------
                  exj=inv(exi)
                  if (key(exj).eq.-1) then
                     goto 50
                  endif
                  key(exj)=-1
                  rclsiz=rclsiz-1
                  if (prev(exj).eq.-1) then
                     start=next(exj)
                     prev(next(exj))=-1
                     goto 50
                  endif
                  if (next(exj).eq.-1) then
                     next(prev(exj))=-1
                     goto 50
                  endif
                  next(prev(exj))=next(exj)
                  prev(next(exj))=prev(exj)
50             continue
               exi=(indx(j)-1)*batch(j)+k*period(j)
40          continue
30       continue
10    continue
      return
      end
```

9.5 Main Driver of the Code

```
c     ----------------------------------------------------------------
c     Program driver for GRASP for the Multidimensional Assignment
c     Problem code
c     Author: Pitsoulis Leonidas
c     ----------------------------------------------------------------
c     This file includes the following Fortran subroutines and
c     functions:
c
c          gmap     - control subroutine for GRASP for MAP algorithm
c          srtcst   - sorts cost
c          stage1   - stage 1 of GRASP construction phase
c          stage2   - stage 2 of GRASP construction phase
c          local    - 2-exchange local search for QAP
c          insrtq   - insert element into heap for sorting
c          removq   - remove element from heap
c          randp    - random number generator function
c
c     -----------------------------------------------------------------
c     The input data consists of the following variables and arrays:
c
c     dim       - the dimension of the array
c     nindex() - the array with the index ranges for each dimension
c                 nindex(1), ... , nindex(dim)
c     c()       - the array elements c(1),...,c(nindex(1)*...*nindex(dim))
c
c     -----------------------------------------------------------------
c     The array c is represented as one-dimensional array c(1)...c(n)
c     where the variable n = nindex(1)*...*nindex(dim).
c     Given and index i in {1...n} we can retrieve the individual
c     indices represented in the variable array indx(dim),
c     indx(1), indx(2), ... , indx(dim) using the subroutine
```

```
c     iindx(--). Given indx(1),...,indx(dim) we can retrieve the
c     index i in {1...n} using the subroutine indxi(--).
c     ---------------------------------------------------------------
      integer nmax,dimmax,npower
      parameter (nmax=22,dimmax=5)
      parameter (npower=15000)
      integer  c(npower)
      integer  srtc(npower),srtic(npower),csrt(npower),icsrt(npower)
      integer  inv(npower),prev(npower),next(npower),key(npower)
      integer  nindex(dimmax),indx(dimmax),indexi(dimmax),indexj(dimmax)
      integer  pm(dimmax,nmax)
      integer  batch(dimmax),numbatch(dimmax),period(dimmax)
      integer  dim,n,nsmall
      integer  seed,objv,iter,i,j,l,totobjv
      real*4   alpha
      real*4   cput(2),bcpu,etime,totcpu
C     ---------------------------------------------------------------
C     Read Problem input data and parameters
C     ---------------------------------------------------------------
      call readp(nmax,dimmax,npower,dim,c,nindex,n,nsmall)
C     ---------------------------------------------------------------
C     Sort the array c() into csrt() to be used and the corresponding
C     indices of csrt() into icsrt()
C     ---------------------------------------------------------------
      call srtcst(n,dim,c,srtc,srtic,csrt,icsrt)
c     ---------------------------------------------------------------
c     Construct the array inv()
c     ---------------------------------------------------------------
      call constinv(n,dim,inv,icsrt)
      call initial(n,dim,dimmax,pm,nmax,objv,batch,numbatch,period,
     +             nindex)

      seed=270001
      alpha=0.1
      iter=100
      objv=0
      totobjv=9999999
c     ---------------------------------------------------------------
c     Main GRASP iterations.
c     ---------------------------------------------------------------
      bcpu=etime(cput)
      do 10 i=1,iter
         call build(n,prev,next,icsrt,key)
         call construct(n,dim,nmax,dimmax,nsmall,c,pm,inv,prev,next,key,
     +              seed,alpha,nindex,indx,batch,numbatch,period,objv)
         call local(n,dim,nmax,dimmax,nsmall,c,pm,nindex,objv,indexi,
     +              indexj)
c         call chkcst(n,dim,nmax,dimmax,pm,c,nindex,costos,nsmall)
         if (objv.lt.totobjv) then
            print *, '####### at iteration : ',i,' #########'
            do 20 l=1,dim
               print *, (pm(l,j), j=1,nindex(l))
20          continue
            print *, '--------------Cost is ---->  ', objv
c            print *, '-------Acutal Cost is ---->  ', costos
```

```
          totobjv=objv
        endif
10    continue
      totcpu=etime(cput)-bcpu
      print *, '---------------------------time---->  ', totcpu
      stop
      end
```

10 Bibliography

References

[1] K. Bassam, P.M. Pardalos, and D.Z. Du. A test problem generator for the steiner problem in graphs. *ACM Transactions on Mathematical Software*, 19(4):509–522, 1993.

[2] T.A. Feo and M.G.C. Resende. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6:109–133, 1995.

[3] J. Hasselberg, P.M. Pardalos, and G. Vairaktarakis. Test case generators and computational results for the maximum clique problem. *Journal of Global Optimization*, 3:463–482, 1993.

[4] B. Korte and C.L. Monma. Some remarks on a classification of oracle-type algorithms. In *Numerische Methoden bei graphentheoretischen und kombinatorischen Problemen, Band 2*, pages 195–215. Birkhauser, Basel, 1979.

[5] Y. Li, P. M. Pardalos, and M. G. C. Resende. Algorithm 754: FORTRAN subroutines for approximate solution of dense quadratic assignment problems using GRASP. *ACM Transactions on Mathematical Software*, 22:104–118, 1996.

[6] Y. Li and P.M. Pardalos. Generating quadratic assignment problems with known optimal permutations. *Computational Optimization and Applications*, 1(2):163–184, 1992.

[7] Y. Li, P.M. Pardalos, and M.G.C. Resende. A greedy randomized adaptive search procedure for the quadratic assignment problem. In P.M. Pardalos and H. Wolkowicz, editors, *Quadratic assignment and related problems*, volume 16 of *DIMACS Series on Discrete Mathematics and Theoretical Computer Science*, pages 237–261. American Mathematical Society, 1994.

[8] T. Mavridou, P. M. Pardalos, L. S. Pitsoulis, and M. G. C. Resende. A GRASP for the biquadratic assignment problem. *European Journal of Operations Research*, 105(3):613–621, 1998.

[9] R. A. Murphey, P. M. Pardalos, and L. Pitsoulis. A greedy randomized adaptive search procedure for the multitarget multisensor tracking problem. In *Network Design: Connectivity and Facility Location*, volume 40 of *DIMACS Series on Discrete Mathematics and Theoretical Computer Science*, pages 277–302. American Mathematical Society, 1997.

[10] P. M. Pardalos, L. S. Pitsoulis, and M. G. C. Resende. A parallel GRASP implementation for solving the quadratic assignment problem. In A. Ferreira and José D.P. Rolim, editors, *Parallel Algorithms for Irregular Problems: State of the Art*, pages 115–133. Kluwer Academic Publishers, 1995.

[11] P. M. Pardalos, L. S. Pitsoulis, and M. G. C. Resende. Algorithm 769: FORTRAN subroutines for approximate solution of sparse quadratic assignment problems. *ACM Transactions on Mathematical Software*, 23:196–208, 1997.

[12] P.M. Pardalos. Generation of large-scale quadratic programs for use as global optimization test problems. *ACM Transactions on Mathematical Software*, 13(2):133–137, 1987.

[13] A. B. Poore and N. Rijavec. Partitioning multiple data sets: multidimensional assignments and lagrangian relaxation. In P.M Pardalos and H. Wolkowicz, editors, *Quadratic assignment and related problems*, volume 16 of *DIMACS Series on Discrete Mathematics and Theoretical Computer Science*, pages 317–342. American Mathematical Society, 1994.

[14] D. J. A. Welsh. *Matroid Theory*. Academic Press, 1976.